PROJECT 3

Minesweeper

Mehmet Emin Atak 2021400282 - Dağhan Erdönmez 2021400093

1. Implementation Details

In this project, we are implementing a custom Minesweeper, including hint and score mechanics in addition to basics of the original game.

Firstly, we worked on the general design of the GUI. We created a MainWindow for our QApplication instance. This MainWindow includes the game itself, which essentially inherits from QWidget. Almost all the properties of the game belong to this QWidget called MinesweeperGame. Additionally, we created a ButtonModified class, in order to customize the QPushButton such that it has a signal for right click button, which we will use to mark cells as flagged. The overall view of the files can be seen below:

In MinesweeperGame, we wrote a function called setupGame() to initialize some matrices and other attributes as needed. This function is used as a slot to be called each time the restart button is clicked. While it creates and refills the matrices “revealed”, “flagged”, “knownMines” and “knownSafes” with “false”, it also creates buttons and puts them into “buttons” matrix. Moreover, it randomly generates the location of mines according to given mine numbers and creates a “mines” matrix to indicate whether a cell has a mine or not.

Note: Matrix means a 2D vector.

Here is the part where we place mines randomly:

```
// Randomly place mines
int minesPlaced = 0;
while (minesPlaced < 10) {
    int row = QRandomGenerator::global()->bounded(rows);
    int col = QRandomGenerator::global()->bounded(cols);
    if (!mines[row][col]) {
        mines[row][col] = true;
        minesPlaced++;
    }
}
```

Then we handle the function cellClicked(), which is the slot to be called when a cell is left clicked. First, it checks whether there is a mine in the clicked cell. If so, it triggers gameOver() to display losing pop-up. Otherwise, it calculates the number of mines in the neighbour cells using countAdjacentMines(x, y) function. If the number obtained from this function is not 0, in other words if there is an adjacent mine for this cell, it resets the icon displayed on the cell to the number by using setIconToButton(). Note that setIconToButton() will be examined later.

In the other case, where there is no mine in adjacent cells, all empty cells adjacent to the cell should be revealed. In order to achieve this, we used a recursive function which iterates through neighbour cells until a non-empty cell is encountered. Here is how it is implemented:

```
void MinesweeperGame::neighbourCells(int row, int col){
    if (visited[row][col] == false) {
        visited[row][col] = true;
        int minesNum = countAdjacentMines(row, col);
        setIconToButton(buttons[row][col], minesNum);
        revealed[row][col] = true;

        if (minesNum == 0) {
            for (int i=-1; i<2; i++) {
                for (int j=-1; j<2; j++) {
                    int newrow = row+i;
                    int newcol = col+j;

                    if (newrow >= 0 && newrow < rows && newcol >= 0 && newcol < cols) {
                        if (i != 0 || j != 0) {
                            neighbourCells(newrow, newcol);

                        }
                    }
                }
            }
        }
    }
}
```

cellClicked() finally checks if the game is won by calling checkWinCondition().

After that, we wrote the function to handle right click. It basically reassigns the boolean on the “flagged” matrix, to the opposite of previous value. So, a flagged cell will be unflagged and an unflagged one will be flagged after the right click.

The game also includes a hint mechanism. When the user clicks the hint button, we need to give an empty cell only if it can be determined to be safe by the user too. In order to achieve this, we used two fundamental rules that each user can think of while trying to solve Minesweeper:

- If the number of a cell’s unrevealed neighbours equals to the number on the cell (number of neighbour with mines), we can be sure that each neighbour is mine.
- If the number of a cell’s neighbour cells that is guaranteed to be mine (by the first rule) equals to the number on the cell, we can be sure that each remaining neighbour is safe.

Firstly, we will turn these rules into functions. Here is the example where we iterate through neighbours and mark them if it is guaranteed to be a mine based on the current knowledge:

```
void MinesweeperGame::markKnownMines(int x, int y) {
    int minesNum = checkAdjacentMines(x, y);
    int unsureCount = 0;
    int unsures[8][2];

    for (int i=-1; i<2; i++) {
        for (int j=-1; j<2; j++) {
            int newx = x + i;
            int newy = y + j;

            if (newx >= 0 && newx < COLS && newy >= 0 && newy < ROWS) {
                if (!revealed[newx][newy] && !knownSafes[newx][newy]) {
                    unsures[unsureCount][0] = newx;
                    unsures[unsureCount][1] = newy;
                    unsureCount++;
                }
            }
        }
    }

    if (minesNum == unsureCount) {
        for (int i=0; i<unsureCount; i++) {
            int nX = unsures[i][0];
            int nY = unsures[i][1];

            knownMines[nX][nY] = true;
        }
    }
}
```

Notice that this function only works for one time. But in practice, a new knowledge of mine or safe may lead to another new knowledge. In order to get this notion of cumulativeness, we created a function called updateKnowns():

```
void MinesweeperGame::updateKnowns() {

    while (true) {
        Matrix oldKnownMines = copyMatrix(knownMines);
        Matrix oldKnownSafes = copyMatrix(knownSafes);

        for (int i=0; i<ROWS; i++) {
            for (int j=0; j<COLS; j++) {
                if (revealed[i][j] && checkAdjacentMines(i, j) != 0) {
                    markKnownMines(i ,j);
                    markKnownSafes(i, j);
                }
            }
        }

        if (matricesAreEqual(oldKnownMines, knownMines) && matricesAreEqual(oldKnownSafes, knownSafes)) {
            break;
        }
    }
}
```

Here it can be seen that our algorithm tries to update knownMines or knownSafes until there is no change in these matrices. Because whenever there is an update (knownMines is changed or knownSafes is changed on this iteration), there is a possibility that we can gather an extra information in the next iteration.

Considering possible efficiency issues, we arranged the hint mechanism like this:

When the hint is clicked, if there is a cell that is not revealed but sured to be safe, give hint. If not, first updateKnowns() and then try again. If still not, then there is no hint based on current user knowledge. Hence, we made sure that updateKnowns() is not called unless it is required. Here is how it looks:

```
void MinesweeperGame::giveHint() {
    int a = canHint();
    if (a != -1) {
        executeHint(a/COLS, a%COLS);
    } else {
        updateKnowns();
        a = canHint();
        if (a != -1) {
            executeHint(a/COLS, a%COLS);
        } else {
            noHint();
        }
    }
}
```

Please note that canHint() is a function that returns the coordinate of hint cell. If not, it returns -1. In order to ease operations, we used x*COLS+y to indicate the coordinate, rather than using x and y values seperately. By a/COLS and a%COLS, we kind of dereference and split it to x and y values again.

The last but one of the most important part was to determine whether the game is won. The basic algorithm behind checkWinCondition() was to iterate through all the cells until it encounters an empty cell that is not revealed yet. If there is one, it immediately returns false. If the iteration goes through all the table, it returns true which means the game is won.

2. UI

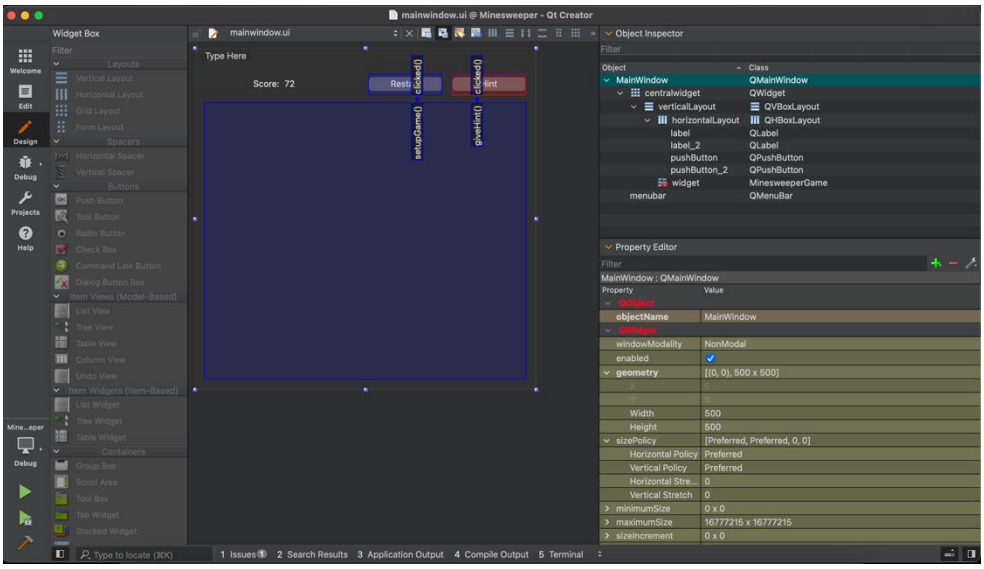
The UI of the game was mainly done using the features of the QT library. We have used QT Creator's WYSIWYG UI editor to easily implement the main UI features.

Buttons and Score:

Using signals and slots, we have connected the score text field to the countRevealed() function.

```
connect(ui->widget, &MinesweeperGame::countRevealed, this, &MainWindow::setScore);
```

This makes the text field update its value every time the value of countRevealed() changes. Which basically corresponds to every time a cell is clicked in the minefield.



Above, you can see the usage of UI designer of QT Creator to connect Restart and Hint buttons to the slots of MinesweeperGame class.

Restart button is connected to setupGame() slot which is basically setting the game from scratch like when the game is first open.

Hint button is connected to the giveHint() slot which starts a long procedure to solve the game to find the mines and give a safe spot as hint. The procedure is described in the implementation details part.

Cell Icons:

Cell icons are implemented using the setIconToButton() function.

Here is a part of the function:

```
void MinesweeperGame::setIconToButton(ButtonModified* button, int minesNum) {
    if (button == NULL) {
        qDebug() << "Button was not found!";
        return;
    }

    switch (minesNum) {
    case -1:
    {
        QPixmap pixmap("../../../../../assets/flag.png");
        pixmap = pixmap.scaled(20, 20);
        QIcon ButtonIcon(pixmap);
        button->setIcon(ButtonIcon);
        button->setIconSize(QSize(20,20));
        break;
```

As you can see it gets a ButtonModified object, which is the custom class that we've implemented, and an integer parameter minesNum. The minesNum parameter is a positive integer between 1-8 for the cells that have mines adjacent to them and a negative number for other possibilities like unrevealed cell, revealed cell with 0 adjacent mines, flagged cells etc.

It basically runs a switch-case for all possibilities. Then assigns the corresponding icon from the assets file to the button as shown. Maybe a notable implementation detail here is numbers. Instead of doing a separate case for each positive number, we have implemented a single default case that uses the minesNum parameter as an argument to find the correct asset. Here is the code:

```
default:
    if (minesNum >= 0 && minesNum <= 8) {
        QPixmap pixmap(QString("../../../../../assets/%1.png").arg(minesNum));
        pixmap = pixmap.scaled(20, 20);
        QIcon ButtonIcon(pixmap);
        button->setIcon(ButtonIcon);
        button->setIconSize(QSize(20,20));
        disconnect(button, &QPushButton::clicked, this, nullptr);
    }
    break;
}
```

3. Challenges Encountered

- QT Creator IDE was a bit unusual for us. UI development with WYSIWYG was useful at first but trying to understand what was really changing when we changed something on the designer was confusing.
- Trying to figure out QT's class structuring, and inheritance method was tough. This was partly because of us not being very familiar with OOP in C++.
- Finding out the logic behind hints was challenging. We were really scared when we figured out that giving hints was not more trivial than solving the game, but it is equivalent to it.